

Evaluating the Efficiency of Asynchronous Systems with FASE*

F. Buti, M. Callisto De Donato, F. Corradini, M.R. Di Berardini

School of Science and Technology, University of Camerino

{federico.buti, massimo.callisto, flavio.corradini, mariarita.diberardini}@unicam.it

W. Vogler

Institut für Informatik, Universität Augsburg

vogler@informatik.uni-Augsburg.de

Abstract

In this paper, we present **FASE** (Faster Asynchronous Systems Evaluation), a tool for evaluating the worst-case efficiency of asynchronous systems. The tool is based on some well-established results in the setting of a timed process algebra (PAFAS: a Process Algebra for Faster Asynchronous Systems). To show the applicability of **FASE** to concrete meaningful examples, we consider three implementations of a bounded buffer and use **FASE** to automatically evaluate their worst-case efficiency. We finally contrast our results with previous ones where the efficiency of the same implementations has already been considered.

1 Introduction

PAFAS [6] has been proposed as a useful tool for comparing the worst-case efficiency of asynchronous systems. It is a CCS-like process description language [10] where basic actions are atomic and instantaneous but have associated a time bound interpreted as the maximal time delay for their execution. These upper time bounds can be used to evaluate efficiency, but they do not influence functionality (which actions are performed); so compared with CCS also PAFAS treats the full functionality of asynchronous systems. In [6], processes are compared via a variant of the testing approach developed by De Nicola and Hennessy in [7]. Tests considered in [6] are test environments (as in [7]) together with a time bound. A process is embedded into the environment (via parallel composition) and satisfies a (timed) test, if success is reached before the time bound in *every* run of the composed system, i.e. even in the worst case. This gives rise to a faster-than preorder over processes that is naturally an *efficiency preorder*. Moreover, this efficiency preorder can be characterised as inclusion of a special kind of *refusal traces*, which provide decidability of the testing preorder for finite state processes.

*This work was supported by the PRIN Project ‘Paco:Performability-Aware Computing: Logics, Models, and Languages’.

In [4], it has been shown that the faster-than preorder provided in [6] can equivalently be defined on the basis of a performance function that gives the worst-case time needed to satisfy any test environment (or user behaviour). If the above timed testing scenario is adapted by considering only test environments that want n tasks to be performed as fast as possible (possibly in parallel), this performance function is *asymptotically linear*. This provides us with a *quantitative* measure of system performance, essentially a function from natural numbers to natural numbers called *response performance function* that measures how fast the system under consideration responds to requests from the environment.

In this paper, we present **FASE**, a corresponding tool that supports the evaluation of this function for a given system. In order to show the applicability of **FASE** to concrete meaningful examples, we consider three different implementations of a bounded buffer and use **FASE** to automatically evaluate their efficiency. The three implementations are called **Fifo**, **Pipe** and **Buff**. **Fifo** is a bounded-length first-in-first-out queue, **Pipe** is a sequence of one place buffers connected end-to-end and **Buff** is an array used in a circular fashion. We prove that **Fifo** is always more efficient than **Pipe** and **Buff**, and that **Buff** is more efficient than **Pipe** only if the number of requests is sufficiently small w.r.t. the size of the buffer. These results are quite different from those presented in [3] (see Section 5) where the efficiency of the same buffer implementations has been compared by means of the efficiency preorder defined in [6]. The reason is that here (as in [4]) we only consider a specific class of user behaviours.

The rest of this paper is organised as follows. Section 2 recalls PAFAS and the technical details we need to define the response performance. Section 3 presents **FASE** and its main algorithms. Section 4 describes the three buffer implementations and states our main results. Finally, Section 5 presents some concluding remarks.

2 PAFAS

In this section we briefly introduce PAFAS, its operational semantics and the performance function to evaluate worst-case efficiency. We refer the reader to [6] and [4] for more details. We use the following notation: \mathbb{A} is an infinite set of basic actions with a special action ω , which is reserved for observers (test processes) in the testing scenario to signal the success of a test. The additional action τ represents an internal activity that is unobservable from other components. Actions in $\mathbb{A}_\tau = \mathbb{A} \cup \{\tau\}$ (ranged over by α, β, \dots) can let time 1 pass before their execution, i.e. 1 is their maximal delay. After that time, they become *urgent* actions. The set of urgent actions is $\underline{\mathbb{A}}_\tau = \{\underline{a} \mid a \in \mathbb{A}\} \cup \tau$ and it is ranged over by $\underline{\alpha}, \underline{\beta}, \dots$. Furthermore, \mathcal{X} is the set of process variables x, y, z, \dots used for recursive definitions. A *general relabelling function* (incorporating relabelling and hiding) is a function $\Phi : \mathbb{A}_\tau \rightarrow \mathbb{A}_\tau$ where the set $\{\alpha \in \mathbb{A}_\tau \mid \emptyset \neq \Phi^{-1}(\alpha) \neq \{\alpha\}\}$ is finite and $\Phi(\tau) = \tau$.

Definition 2.1 (*Timed Processes*) The set \mathbb{P} of (*timed*) *processes* is the set of closed (i.e. without free variables) and guarded (i.e. variable x in a $\mu x.P$ only appears within the scope of a prefix $\alpha.(.)$, where $\alpha \in \mathbb{A}_\tau$) terms generated by the following grammar:

$$P ::= 0 \mid \gamma.P \mid P + P \mid P \parallel_A P \mid P[\Phi] \mid x \mid \mu x.P$$

where γ is α or $\underline{\alpha}$ for some $\alpha \in \mathbb{A}_\tau$, Φ a general relabelling function, $x \in \mathcal{X}$ and $A \subseteq \mathbb{A}$ possibly infinite.

A brief description of our operators now follows. 0 is the Nil-process, which cannot perform any action, but may let time pass without limit¹; $\alpha.P$ and $\underline{\alpha}.P$ is (action-) prefixing, known from CCS. In particular, process $a.P$ performs a with a *maximal* delay of 1; hence, it can either perform a immediately, or can idle for time 1 and become $\underline{a}.P$. In the latter case, the idle-time has elapsed and action a must either occur or be deactivated (in a choice-context) before time may pass further. Our processes are *patient*: as a stand-alone process, $\underline{a}.P$ has no reason to wait; but as a component in $\underline{a}.P \parallel_{\{a\}} a.Q$, it has to wait for synchronisation on a and this can take up to time 1, since the component $a.Q$ may idle this long. $P_1 + P_2$ models the choice between two conflicting processes P_1 and P_2 . $P_1 \parallel_A P_2$ is the TCSP-like parallel composition of two processes P_1 and P_2 that run in parallel and have to synchronise on all actions from A [2]. In the following we write \parallel as a shorthand for $\parallel_{\mathbb{A} \setminus \{\omega\}}$. $P[\Phi]$ behaves as P but with the actions changed according to Φ . Finally, $\mu x.P$ models a recursive definition; recursive equations are a common way of defining processes.

We now define the refusal traces of a process P . Intuitively, a refusal trace records, along a computation, which actions P can perform ($P \xrightarrow{\alpha}_r P'$, $\alpha \in \mathbb{A}_\tau$) and which actions P can refuse to perform ($P \xrightarrow{X}_r P'$, $X \subseteq \mathbb{A}$). A transition like $P \xrightarrow{X}_r P'$ is called a (*conditional*) *time step*. The actions in the set X are not urgent (see rule Pref_{r2} in Fig. 1) so P is justified in not performing them but performing a time step instead. Since other actions might be urgent and cannot be refused, P as a stand-alone-process might actually be unable to let time pass. But if P is a component of a larger system, these actions might be further delayed due to synchronisation with some other components, and a time step is possible. Whenever P can make a time step in *any* context (i.e. if $P \xrightarrow{X}_r P'$ and $X = \mathbb{A}$), we say that P performs a *full time step* and also write $P \xrightarrow{1}_r P'$.

Definition 2.2 (*Refusal operational semantics*) The SOS-rules in Fig. 1 (plus symmetric rules for Par_{a1} and Sum_a for actions of P_2) define the transition relations $\xrightarrow{\alpha}_r \subseteq (\mathbb{P} \times \mathbb{P})$ for $\alpha \in \mathbb{A}_\tau$ and $\xrightarrow{X}_r \subseteq (\mathbb{P} \times \mathbb{P})$ for $X \subseteq \mathbb{A}$.

The rules in Fig. 1 explain the operational semantics of PAFAS processes. A process like $\alpha.P$ can either perform action α immediately and then become P (rule Pref_{a1}), or can let time 1 pass and refuse any set of actions (rule Pref_{r1}). A process $\underline{\alpha}.P$ can perform an action α (rule Pref_{a2}) and on its own cannot delay such an execution (rule Pref_{r2}). Since internal action τ has never to be synchronised, a process prefixed by an urgent τ cannot make a time step. Another rule worth noting is Par_r that defines which actions a parallel composition can refuse during a time step. The intuition is that $P_1 \parallel_A P_2$ can refuse an action a if either $a \notin A$ (P_1, P_2 are not forced to synchronise on a) and both P_1, P_2 can refuse a , or $a \in A$ (P_1, P_2 are forced to synchronise on a) and either P_1 or P_2 can refuse a . The other rules are as expected.

For sequences $w \in (\mathbb{A}_\tau \cup 2^{\mathbb{A}})^*$, we define $P \xrightarrow{w}_r P'$ as expected: $P \xrightarrow{w}_r P'$ if either $w = \varepsilon$ (the empty sequence) and $P' = P$ or there is $Q \in \mathbb{P}$ and $\mu \in (\mathbb{A}_\tau \cup 2^{\mathbb{A}})$ such that $P \xrightarrow{\mu}_r Q \xrightarrow{w'}_r P'$ and $w = \mu w'$. Similarly, we define $P \xrightarrow{w}_r P'$ for $w \in (\mathbb{A}_\tau \cup \{1\})^*$. In the latter case, $\zeta(w)$ is the duration of w , i.e. the number of full time steps in w . We write $P \xrightarrow{v}_r P'$ ($P \xRightarrow{v}_r P'$) if $P \xrightarrow{w}_r P'$ ($P \xrightarrow{w}_r P'$, resp.) and $v = w/\tau$ (v is the sequence w with all τ 's removed). Finally, $\text{RT}(P) = \{w \mid P \xrightarrow{w}_r\}$ and $\text{DL}(P) = \{w \mid P \xRightarrow{w}_r\}$ are the sets of *refusal traces* and *discrete traces* (resp.) of P .

¹A trailing 0 will often be omitted, so e.g. $a.b + c$ abbreviates $a.b.0 + c.0$.

$$\begin{array}{c}
\text{PREF}_{a1} \frac{}{\alpha.P \xrightarrow{r} P} \quad \text{PREF}_{a2} \frac{}{\underline{\alpha}.P \xrightarrow{r} P} \quad \text{SUM}_a \frac{P_1 \xrightarrow{r} P'_1}{P_1 + P_2 \xrightarrow{r} P'_1} \\
\\
\text{PAR}_{a1} \frac{\alpha \notin A, P_1 \xrightarrow{r} P'_1}{P_1 \|_A P_2 \xrightarrow{r} P'_1 \|_A P_2} \quad \text{PAR}_{a2} \frac{\alpha \in A, P_1 \xrightarrow{r} P'_1, P_2 \xrightarrow{r} P'_2}{P_1 \|_A P_2 \xrightarrow{r} P'_1 \|_A P'_2} \\
\\
\text{REL}_a \frac{P \xrightarrow{r} P'}{P[\Phi] \xrightarrow{\Phi(\alpha)} P'[\Phi]} \quad \text{REC}_a \frac{P\{\mu x.P/x\} \xrightarrow{r} P'}{\mu x.P \xrightarrow{r} P'} \\
\\
\text{NIL}_r \frac{}{0 \xrightarrow{r} 0} \quad \text{PREF}_{r1} \frac{}{\alpha.P \xrightarrow{r} \underline{\alpha}.P} \quad \text{PREF}_{r2} \frac{\alpha \notin X \cup \{\tau\}}{\underline{\alpha}.P \xrightarrow{r} \underline{\alpha}.P} \\
\\
\text{PAR}_r \frac{\forall_{i=1,2} P_i \xrightarrow{X_i} P'_i, X \subseteq (A \cap \bigcup_{i=1,2} X_i) \cup ((\bigcap_{i=1,2} X_i) \setminus A)}{P_1 \|_A P_2 \xrightarrow{X} P'_1 \|_A P'_2} \\
\\
\text{SUM}_r \frac{\forall_{i=1,2} P_i \xrightarrow{X} P'_i}{P_1 + P_2 \xrightarrow{X} P'_1 + P'_2} \quad \text{REL}_r \frac{P \xrightarrow{\Phi^{-1}(X \cup \{\tau\}) \setminus \{\tau\}} P'}{P[\Phi] \xrightarrow{X} P'[\Phi]} \\
\\
\text{REC}_r \frac{P\{\mu x.P/x\} \xrightarrow{X} P'}{\mu x.P \xrightarrow{X} P'\{\mu x.P/x\}}
\end{array}$$

Figure 1: The Refusal Operational Semantics of PAFAS processes.

For processes $P, Q \in \mathbb{P}$, $\text{RT}(P) \subseteq \text{RT}(Q)$ implies $\text{DL}(P) \subseteq \text{DL}(Q)$: $\text{DL}(P)$ corresponds to the set of traces $w \in \text{RT}(P)$ where $X = \mathbb{A}$ for all refusal sets X in w . Finally, the *refusal transition system* $\text{RTS}(P)$ of P is defined as the set of all transitions $Q \xrightarrow{\mu} Q'$ with $\mu \in \mathbb{A}_r$ or $\mu \subseteq \mathbb{A}$ where Q is reachable from P via such transitions. It is easy to prove that $\text{RTS}(P \|_A Q)$ can be determined from $\text{RTS}(P)$ and $\text{RTS}(Q)$ according to the SOS-rules for parallel composition given in Fig. 1.

In the timed testing of [6], P satisfies a timed test (observer O with special success action ω plus time bound D) if every discrete trace of $P \| O$ performs ω before time D ; P is *faster than* Q , $P \sqsupseteq Q$, if P satisfies all timed tests that Q satisfies. This preorder is a qualitative notion since a timed test is either satisfied or not, and a process is more efficient than another or not.

One of the main results in [6] is that the faster-than preorder can be characterised by refusal-trace-inclusion, i.e. $P \sqsupseteq P'$ iff $\text{RT}(P) \subseteq \text{RT}(P')$ (see Theorem 5.13 in [6]). A new formulation of this preorder has been provided in [4] (see Prp. 9) that brings to light its quantitative nature; the new formulation is given using the following performance function:

In [4], Prop. 9 provides

Definition 2.3 (*Performance*) Let $P \in \mathbb{P}$ be a process and $O \in \mathbb{P}$ be a test process. We define the *performance function* p as:

$$p(P, O) = \sup\{n \in \mathbb{N}_0 \mid \exists v \in \text{DL}(P \| O) : \zeta(v) = n \text{ and } v \text{ does not contain } \omega\}$$

If the right-hand side has no maximum, the supremum is ∞ . The performance function p_P is defined by $p_P(O) = p(P, O)$, and we write $P \sqsupseteq Q$ if $p_P(O) \leq p_Q(O)$ for each O .

The performance function p (as well as the preorder \sqsupseteq) contrasts processes w.r.t. all possible test environments. In some cases, this might be too demanding and one can make some reasonable assumption about the user behaviour. Consider a scenario where users have a number of requests (made via *in*-actions) that they want to be answered (via *out*-actions) as fast as possible. This class of users is defined as $\mathcal{U} = \{U_n \mid n \geq 1\}$ where $U_1 \equiv \underline{in.out.\omega}$ and $U_n = U_{n-1} \parallel_{\{\omega\}} \underline{in.out.\omega}$ (for any $n > 1$). Given these users, we can define the *response performance* rp of a testable process P as a function from \mathbb{N} to \mathbb{N}_0 with $rp_P(n) = p_P(U_n) = p(P, U_n)$; here n is the size (i.e. the number of requests) of the user.

In what follows we briefly describe how the response performance of a process P can be calculated from its refusal transition system. We restrict attention to so-called response processes, which never produce an *out* without a corresponding preceding *in*.

By Definition 2.3, to determine $rp_P(n)$ we have to consider all $w \in \text{DL}(P \parallel U_n)$ that do not contain ω , count the number of their full time steps and then take the supremum of the numbers so obtained. These traces are just paths in $\text{RTS}(P \parallel U_n)$ that do not contain ω and contain only full time steps. These paths can have at most n *in*'s and n *out*'s (due to synchronisation with U_n). But after the n -th *out*, an urgent ω becomes available and no more full time steps can occur before ω ; in other words, full time steps are only possible before the n -th *out*. So we have solely to consider paths in $\text{RTS}(P \parallel U_n)$ that contain only full time steps and have at most n *in*'s and $(n - 1)$ *out*'s (and, hence, no ω). In [4] it has been proven that for each of these paths there is a so-called n -critical path in $\text{rRTS}(P)^2$ with the same number of time steps. Thus, the following characterisation for the response performance can be given.

Theorem 2.4 (*Characterisation for response performance*) A path in $\text{rRTS}(P)$ is n -critical if it contains at most n *in*'s, at most $n - 1$ *out*'s, and all time steps before the n -th *in* are full. The response performance of a process P is the supremum of the numbers of time steps taken over all n -critical paths.

Now a key observation is that, when the number n of requests is large compared to the number of processes in $\text{rRTS}(P)$, an n -critical path with many time steps must contain cycles. Finding the worst cycles turns out to be essential for performance evaluation. In [4], these worst cycles are distinguished to be either *catastrophic* or *bad* cycles.

Definition 2.5 (*Catastrophic cycle*) A cycle in $\text{rRTS}(P)$ is a catastrophic cycle if it has a positive number of time steps but no *in*'s and no *out*'s. If $\text{rRTS}(P)$ has a catastrophic cycle then $rp_P(n) = \infty$ for some n .

Intuitively, once in a catastrophic cycle, we cannot satisfy any other request (this is because a catastrophic cycle does not contain *out*-actions) but time can pass indefinitely (the cycle has at least one time step). As a consequence, there exists some n (depending on how many *in* and *out*-actions are performed on a path in $\text{rRTS}(P)$ from P to this cycle) such that $rp_P(n) = \infty$, i.e. some user is not satisfied within a bounded time. If $\text{rRTS}(P)$ is free from catastrophic cycles we search for the so called bad cycles:

²This is a reduced version of the $\text{RTS}(P)$ where all conditional time steps, that cannot participate in a full time step when P runs in parallel with a user U_n , are removed. For more details see [4].

Definition 2.6 (*Bad cycle*) For P without catastrophic cycles, we consider cycles reached from P by a path where all time steps are full and which themselves contain only full time steps. Let the *average performance* of such a cycle be the ratio between the number of its full time steps and the number of its *in* actions. A *bad cycle* is a cycle in $\text{rRTS}(P)$ which has maximal average performance.

Theorem 16 in [4] shows that rp_P is asymptotically linear, i.e. $\exists a \in \mathbb{R}$ s.t. $rp_P(n) = an + \Theta(1)$, and that the “asymptotic performance” a of P is the average performance of a bad cycle. In other words, while n -critical paths give the exact value of the response performance of a process, the average performance of a bad cycle is its asymptotic behaviour. Both catastrophic and bad cycles can be automatically checked with FASE.

3 Performance evaluation with FASE

In this section we introduce FASE³, the tool that has been used to automatically evaluate the worst-case efficiency of the three buffer implementations discussed in Section 4. FASE is written in Java language and consists of two main components. The former one is essentially a parser unit; it takes as input a sequence of characters that represents a PAFAS process P and builds its $\text{RTS}(P)$. The second one is the performance module that implements the algorithms used to evaluate the worst-case efficiency of P . The two modules are loosely coupled; they communicate via a shared Java data structure or via an XML-based representation of the RTS . The last aspect is very important since changes to a module do not affect the other one; moreover, the XML interface guarantees a broader interoperability with external tools such as graph visualisers, which could be useful for further analysis of the modelled systems.

3.0.1 Parsing unit

Fig. 2 shows on top the parsing phase that is based on two well-known tools: JFlex [9] as the lexer generator and jacc [11] as the parser generator. JFlex defines how input streams must be arranged into words - called *tokens* - while jacc pseudocode gives rules - called *productions* - to compound such tokens. These productions are used by the parser to generate the data structure that contains the hierarchical representation of the process where each element is a term of the grammar in Definition 2.1. For example, after parsing $P = a.nil + \underline{b}.nil$, the hierarchy structure obtained has on top the process variable P which contains a choice operator with a prefix $a.nil$ and an urgent prefix $\underline{b}.nil$ respectively, and so on. Every element is an instance of a Java class that handles the respective SOS rules given in Fig. 1; thus, an element encapsulates both functional and temporal behaviour used to generate $\text{RTS}(P)$ as indicated at the bottom of Fig. 2.

The building process of $\text{RTS}(P)$ exploits the hierarchical structure, traversing it from the root element; at each step the operator objects generate the proper nodes and transitions according to Definition 2.2. For instance, $P = a.nil + \underline{b}.nil$ will produce the node P with two outgoing transition a and b to the same node nil ; the additional refusal transition $\{a\}$ to the process $\underline{a}.nil + \underline{b}.nil$ will be produced according to rules SUM_r , PREF_{r1} and PREF_{r2} . The same method will be applied to the remaining nodes as expected.

³<http://cosy.cs.unicam.it/fase/>

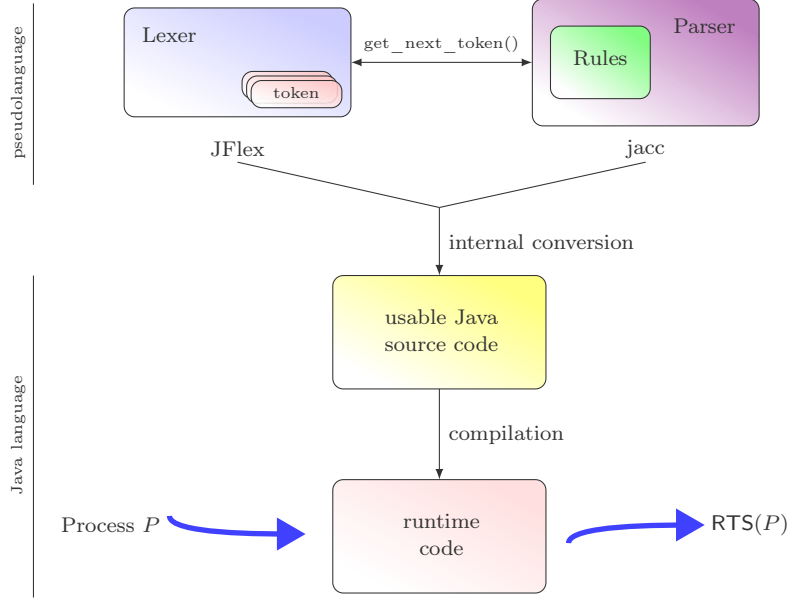


Figure 2: An architectural overview of FASE.

Such an architectural structure provides several advantages. The pseudocode of both lexer and parser are based on common syntaxes (such as regular expressions and BNF rules) that are extremely smaller than actual Java code, easier to understand and easier to maintain. Semantics of each operator is coded in a separate compile unit, hence it can be specified independently and modified in a second stage, if necessary.

3.0.2 Performance unit

The performance component provides all the algorithms needed to evaluate systems performance according to the theoretical results stated in the previous section. In particular, FASE adopts two new algorithms for catastrophic cycles detection and bad cycle calculation that improve those proposed in [4]. Moreover, FASE is also able to generate the complete set of traces that characterises the behaviour of the process. Such diagnostic information is useful to the user since it helps to understand why a modelled system produces catastrophic cycles or has certain worst-case performance. This feature has helped us to validate the results on the response performance of the three buffer implementations discussed in the next section.

Catastrophic cycles The problem of finding catastrophic cycles in a process P has been solved in [4] in time $\theta(N^3)$ where N is the number of nodes in $\mathbf{rRTS}(P)$. The new algorithm adopted in FASE takes advantage of the well-known problem of finding the *Strongly Connected Components* (SCCs) [1]. Since an SCC of a graph is a subgraph that is strongly connected and maximal, the following suffices. We obtain a new graph G from $\mathbf{rRTS}(P)$ by deleting all edges labelled *in* and *out* and apply the algorithm for finding the SCCs. If at least one contains some time step, we can conclude that P has a catastrophic cycle. Indeed, if S is an SCC in G and there is a time step (u, v) with u and v nodes of S , then S has a path from v to u , i.e. it has a catastrophic cycle that is also contained in $\mathbf{rRTS}(P)$. Vice versa, if P has a catastrophic cycle, it is contained in some SCC of G , which therefore contains a time step.

<i>Cells</i>	<i>nodes/edges</i>	<i>previous and</i>	<i>new algorithm</i>	<i>Gain</i>
Pipe ₅	114/292	37	16	74.1%
Buff ₅	96/216	15	15	—
Pipe ₆	272/759	578	63	89.1 %
Buff ₆	160/368	93	22	76.3 %
Pipe ₇	648/1958	11484	296	97.4 %
Buff ₇	240/560	390	47	87.9 %
Pipe ₈	1544/5034	620109	1575	99.7%
Buff ₈	336/792	1172	70	94.0 %
Pipe ₉	3680/12902	—	9687	100 %
Buff ₉	448/1064	2922	109	96.2%

Table 1: Catastrophic-cycle detection time (expressed in *ms*)

The standard SCC discovery algorithm has complexity $O(N + E)$ with N and E the number of nodes and edges of G respectively, and the same applies to construction of G and thus to finding catastrophic cycles in FASE. Table 1⁴ reports the running time for the original and the new algorithm.

Bad cycles Next we look for a bad cycle, possibly not unique, of $\text{rRTS}(P)$ according to Definition 2.6 that gives the average performance of P . To determine this value, a graph G is obtained from $\text{rRTS}(P)$ by deleting all non-full time steps and all nodes not reachable any more (see Proof of Theorem 17 of [4] for more details). To apply a known algorithm from the literature, we do not look for a cycle with maximal average performance in G , but for one with minimal average throughput, the latter being just the inverse of the average performance. Such a cycle can be seen as a set of sub-paths where each one ends in a time step.

For the known algorithm, we must transform G to a graph G' where each edge is weighted with some cost and represents one time step, i.e. an edge corresponds to such a sub-path. Since the costs should be minimal, the subpath without the last node must be a shortest path between the respective nodes as measured by the number of *in*'s. Hence, one obtains a new graph G_0 by deleting all time steps in G and computes its all-pairs shortest paths matrix d with the Floyd-Warshall algorithm, considering a weight 1 for *in*-transitions and 0 for all the other edges. The final G' graph is constructed from the nodes of G on the basis of the matrix d ; for every two nodes u, v of G , where $d(u, v)$ is finite and there exists a time step from v to v' , we add the edge (u, v') with cost $d(u, v)$. This construction can be carried out in time $O(N^3)$. Now the problem of finding the minimal average throughput t can be solved with Karp's algorithm [8] applied to graph G' .

Although the above method is bounded by a complexity of $O(N^3)$, the construction of the shortest-paths matrix d has a cost of $\Theta(N^3)$. In a common scenario where the behaviour of P can be very complex, the computation of the matrix could be expensive as reported in Table 2. To get around the problem, we have developed an improved algorithm. Starting from G and G_0 as defined above, we reverse the edges of G_0 to obtain the graph G_0^T . Since we are only interested in paths leading to a time step, for each full time step (v, v') of G , we apply Dijkstra's

⁴Pipe and Buff are two different implementation of the same buffer discussed in the next section. We have left out Fifo since its representation is too small for sensible comparison.

<i>Cells</i>	<i>nodes/edges of G</i>	<i>previous and</i>	<i>new algorithm</i>	<i>nodes/edges of G'</i>	<i>Gain</i>
Pipe ₅	114/292	546	62	114/3648	88.6 %
Buff ₅	96/216	469	62	96/4608	86.7 %
Pipe ₆	272/759	4279	266	272/17408	93.7 %
Buff ₆	160/368	1422	172	160/12800	87.9 %
Pipe ₇	648/1958	15000	1438	648/82944	90.4 %
Buff ₇	240/560	7485	437	240/28800	94.1 %
Pipe ₈	1544/5034	—	6672	1544/395264	100 %
Buff ₈	336/792	12454	734	336/56448	94.1 %
Pipe ₉	3680/12648	—	56000	3680/1884160	100 %
Buff ₉	448/1064	45031	1766	448/100352	96.0 %

Table 2: Construction time of G' (expressed in *ms*)

algorithm to G_0^T with root node v and weight 1 for *in*-transitions, 0 otherwise as above. Finally, for each node u , such that there exists a path from v , we add an edge (u, v') in G' where the cost is the length of a shortest path from v to u .

With this approach, we calculate only those (shortest) paths that lead to time steps, i.e. only those paths that correspond to edges in G' . On the contrary, in the original algorithm, (shortest) paths between all pairs of nodes are computed. Since Dijkstra’s algorithm runs in time $O(E + N \log N)$ [1] (with N and E the number of nodes and edges respectively), constructing G' takes $O(N(E + N \log N))$, but at least the first factor N will be considerably smaller in practice. Table 2 shows the improvements obtained when considering large buffer implementations.

4 Evaluating the performance of three bounded buffer implementations

In this section, we evaluate the worst-case efficiency of three implementations of a bounded buffer (of capacity $N + 2$, where $N \geq 1$ is a fixed natural number) with FASE. These implementations have already been considered in [3] where their efficiency has been compared via the faster-than preorder relation \sqsubseteq defined in [6]. In particular, we want to investigate if the results stated in [3] still hold in our quantitative setting with the restricted class of users. The three implementations are **Fifo** (a bounded-length first-in-first-out queue), **Pipe** (a sequence of one place buffers connected end to end) and **Buff** (an array used in a circular fashion). Unlike [3], we abstract away from the actual values stored in the buffers and assume that the latter perform, as visible actions, either an *in*-action (meaning that a value is saved in a free cell of the buffer) or an *out*-action (meaning that the buffer gives back a value to the external environment). This choice surely does not influence performance as already discussed in [4], since the operations are data-independent, and it allows us to reduce considerably the number of states considered when calculating the response performance.

The first buffer implementation **Fifo** shown in Fig. 3 directly implements a first-in-first-out queue of capacity $N + 2$. It has no overhead in terms of internal actions and it is purely sequential. In the examples, we use names and defining equations (using \equiv) to describe recursive behaviour.



Figure 3: The Software Architecture for Fifo

Definition 4.1 (*The buffer Fifo*) We define $\text{Fifo} \equiv \text{Fifo}(0)$ where, for each $i = 0, \dots, N + 2$, $\text{Fifo}(i)$ is defined as follows:

1. $\text{Fifo}(0) \equiv \text{in}.\text{Fifo}(1)$
2. for $0 < i < N + 2$ then $\text{Fifo}(i) \equiv \text{in}.\text{Fifo}(i + 1) + \text{out}.\text{Fifo}(i - 1)$
3. $\text{Fifo}(N + 2) \equiv \text{out}.\text{Fifo}(N + 1)$

Proposition 4.2 *The asymptotic performance of Fifo is 2 (i.e. $rp_{\text{Fifo}}(n) = 2n + \Theta(1)$). Moreover, for any $N \geq 1$, $rp_{\text{Fifo}}(n) = 2n$.*

Proof: We have used FASE in order to automatically prove that Fifo does not have catastrophic cycles and to calculate its asymptotic performance. For what concerns its response performance, we can easily see that Fifo may need a time step for any input or output. E.g. $(\mathbb{A} \text{ in } \mathbb{A} \text{ out})^{n-1} \mathbb{A} \text{ in } \mathbb{A}$ is an n -critical path with a maximum number of time steps. We can conclude that $rp_{\text{Fifo}}(n) = 2n$. \square

A buffer can also be implemented as a concatenation of $N + 2$ cells as shown in Fig. 4, where a cell is an input/output device that contains at most one value. In such a case, the cells have to be connected end-to-end, so that the output of each cell becomes the input of the next one.

Definition 4.3 (*The buffer Pipe*) We define an empty cell as the process $C \equiv \text{in}.C'$ where $C' \equiv \text{out}.C$. Let $i = 0, \dots, N + 1$; the i -th cell of Pipe is defined by $C_i \equiv C[\Phi_i]$ where the relabelling function Φ_i is defined as follows:

$$\Phi_i(\alpha) = \begin{cases} \delta_i & \text{if } \alpha = \text{in} \text{ and } 0 \leq i \leq N \\ \delta_{i-1} & \text{if } \alpha = \text{out} \text{ and } 1 \leq i \leq N + 1 \\ \alpha & \text{otherwise} \end{cases}$$

Here each action δ_j passes the value from the $(j + 1)$ -th to the j -th cell. We force synchronisation among two consecutive cells by properly relabelling in and out-actions of single cells. Let $A = \{\delta_0, \delta_1, \dots, \delta_{N+1}\}$. We define $\text{Pipe} \equiv (C_0 \parallel_{\delta_0} C_1 \parallel_{\delta_1} \dots \parallel_{\delta_{N+1}} C_{N+1})/A$ where, for any given $P \in \mathbb{P}$, the process P/A is the same as $P[\Phi_A]$ where $\Phi_A(\alpha) = \tau$ if $\alpha \in A$ and $\Phi_A(\alpha) = \alpha$ if $\alpha \notin A$.

Besides input and output of values, Pipe performs a number of activities in order to manage the queue of cells, i.e. to move values from a cell to the next one. These actions are synchronisations between consecutive cells on actions δ_i and have been made internal. Moreover, note that Pipe receives input values in cell $N + 1$ (the only *in*-action not renamed by functions Φ_i is the one performed by this cell) and delivers output values at cell 0.

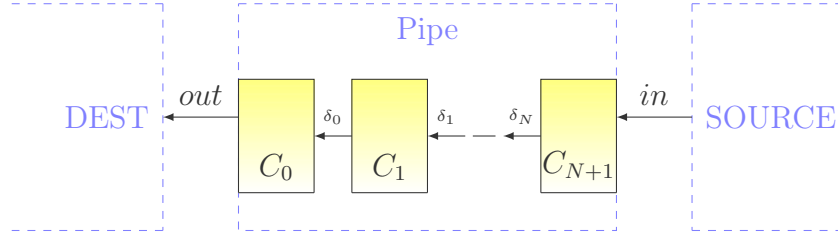


Figure 4: The Software Architecture for Pipe

Proposition 4.4 *The asymptotic performance of Pipe is 2. Moreover, for any $N \geq 1$, we have that $rp_{\text{Pipe}}(n) = 2n + (N + 1)$.*

Sketch of the proof: Again, we have used FASE to prove that $\text{rRTS}(\text{Pipe})$ does not contain catastrophic cycles and to evaluate Pipe's asymptotic performance. We only indicate why $rp_{\text{Pipe}}(n) = 2n + (N + 1)$. The first value is moved to cell $N + 1$ after one time step; with every further time step, it is moved to the next cell; so it arrives in cell 0 after $N + 2$ time steps and is delivered with the next one. After the second time step, cell $N + 1$ becomes empty, so the second value is put into cell $N + 1$ after three time steps and then moves along the pipe with the same speed as the first one. Thus, the next value is always delivered after two more time steps; see [5] for the formal treatment of a more general case. \square

In Fig. 5 it has been assumed that N cells are not connected end-to-end but are used as a storage. These cells interact with a centralised buffer controller which can store two more values and uses the cells in the storage as a circular queue (ordered as $0 < 1 < \dots < N - 1$). In this case, it is the buffer controller that interacts with the external environment. More in detail, the buffer controller accepts a value from the external environment and then writes it in the first empty cell. It also reads the oldest undelivered value from the array and outputs it whenever possible. In the following we write $a \oplus b$ to denote $(a + b) \bmod N$.

Definition 4.5 (*The buffer Buff*) *Let $i = 0 \dots N - 1$. The i -th cell of the storage is described by the process $B_i \equiv C[\Phi'_i]$ where the relabelling functions Φ'_i are defined by*

$$\Phi'_i(\alpha) = \begin{cases} \omega_i & \text{if } \alpha = in \\ \rho_i & \text{if } \alpha = out \\ \alpha & \text{otherwise} \end{cases}$$

Here we use the action ω_i (ρ_i) to denote the writing of a value into the storage (the reading of a value from the storage, respectively). Let $B = \{\omega_j, \rho_j \mid i = 0, \dots, N\}$ be the set of all these actions and $\text{Mem} \equiv (B_0 \parallel_{\emptyset} \dots \parallel_{\emptyset} B_{N-1})$.

The state of the buffer controller, $\text{BC}(x, y, i, m)$, is determined by four arguments: $x, y \in V = \{\perp, \square\}$ are used to represent the absence or presence of an input value (output value resp.) (see below) stored in BC , i is the index of the cell that contains the oldest undelivered value and m is the number of values currently stored in Mem . If $x = \perp$ the buffer controller can accept a new value, otherwise (i.e. if $x = \square$) it has to wait until the last accepted value is actually stored in Mem . Analogously, if $y = \square$, then the buffer controller is ready to produce an output and if $y = \perp$ no value is available for immediate output. Let $x, y \in V$, $0 \leq i \leq N - 1$ and $0 \leq m \leq N$. We define:

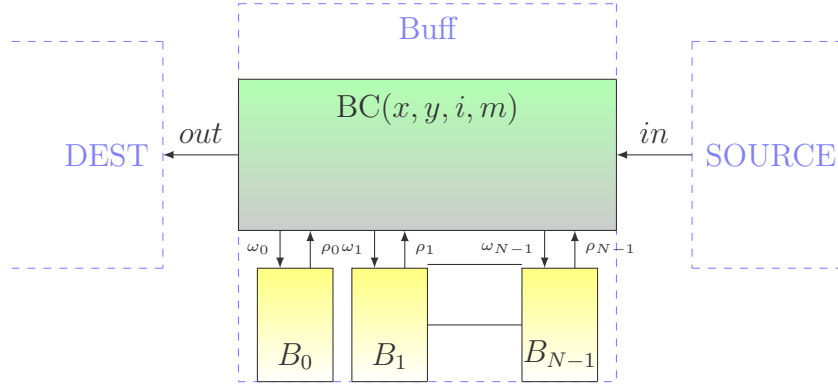


Figure 5: The Software Architecture for Buff

1. $BC(\perp, \perp, i, 0) \equiv in.BC(\square, \perp, i, 0);$
2. $m > 0$ implies $BC(\perp, \perp, i, m) \equiv in.BC(\square, \perp, i, m) + \rho_i.BC(\perp, \square, i \oplus 1, m - 1);$
3. $BC(\square, \perp, i, 0) \equiv \omega_i.BC(\perp, \perp, i, 1);$
4. $0 < m < N$ implies $BC(\square, \perp, i, m) \equiv \omega_{i \oplus m}.BC(\perp, \perp, i, m + 1) + \rho_i.BC(\square, \square, i \oplus 1, m - 1);$
5. $BC(\square, \perp, i, N) \equiv \rho_i.BC(\square, \square, i \oplus 1, N - 1);$
6. $BC(\perp, \square, i, m) \equiv in.BC(\square, \square, i, m) + out.BC(\perp, \perp, i, m);$
7. $m < N$ implies $BC(\square, \square, i, m) \equiv \omega_{i \oplus m}.BC(\perp, \square, i, m + 1) + out.BC(\square, \perp, i, m);$
8. $BC(\square, \square, i, N) \equiv out.BC(\square, \perp, i, N).$

We finally define $\text{Buff} \equiv (\text{Mem} \parallel_B BC(\perp, \perp, 0, 0))/B$. Notice that in such a case all the actions we use to read and write values in **Mem** are made internal.

Proposition 4.6 For any $N \geq 1$, we have $rp_{\text{Buff}}(n) = 4n$.

Proof: Also in this case we have used FASE to prove that $\text{rRTS}(\text{Buff})$ does not have catastrophic cycles and to evaluate its asymptotic performance. Concerning its response performance, consider first the case of one value: after a time step, the value is taken into the input part of BC; after another time step, it is moved into **Mem**; after the third time step it is moved into the output part of BC; after the fourth time step, it is delivered. For several values, these sequences can be interleaved to some degree; but since BC takes part in each action, all these actions are performed sequentially, and always after a time step in the worst case. E.g. for $n = kN + m$ for some $k \geq 1$ and $m < N$, first we fill up and clear the buffer with the sequence $((\mathbb{A} in \mathbb{A} \tau)^N (\mathbb{A} \tau \mathbb{A} out)^N)^k$, fill it up again with a sequence $(\mathbb{A} in \mathbb{A} \tau)^m$ and finally empty it with the sequence $(\mathbb{A} \tau \mathbb{A} out)^{m-1} \mathbb{A} \tau \mathbb{A}$. All paths in that form (up to permutations) are n -critical paths with the maximum number of time steps that is $4Nk + 2m + 2(m - 1) + 2 = 4n$. \square

Now we can state the main result of this paper. This follows as a straightforward consequence of Propositions 4.2, 4.4 and 4.6.

Corollary 4.7 For any $N \geq 1$, *Fifo* is more efficient than both *Pipe* and *Buff* (w.r.t. the quantitative point of view). Moreover, *Buff* is more efficient than *Pipe* iff $n \leq \lfloor \frac{N+1}{2} \rfloor$.

5 Concluding remarks

The results obtained with our tool are quite different from those presented in [3] where the same buffer implementations have been compared using the efficiency preorder defined in [6]. In [3] it is stated that **Fifo** and **Pipe** are unrelated according to the worst-case efficiency preorder (unrelated means that the former process is not more efficient than the second one and vice versa). Similarly **Buff** and **Pipe** are unrelated. The authors provide good reasons for these results and also prove that **Fifo** is more efficient than **Buff** but not vice versa.

As already stated in the introduction, the efficiency preorder is based on arbitrary test environments, whereas we have only used restricted environments adequate for quantitative reasoning in this paper. To explain the results of [3], we consider the refusal trace $v = in \mathbb{A} \emptyset out \{in\} \in RT(\text{Fifo}) \setminus RT(\text{Pipe})$, which can be understood as a witness of slow behaviour of **Fifo**, justifying $\text{Fifo} \not\sqsubseteq \text{Pipe}$. This trace tells us that **Fifo** can perform two time steps after an *in* provided the environment does not offer a communication after the first one (**Fifo** itself would neither block *in* nor *out*); then it can deliver the value and can now delay *in* (as after any visible action). Now we show that none of our users can be such a suitable context, i.e. that **Fifo** cannot participate in such a discrete trace v when running in parallel with a user U_n ; hence, v is not relevant for ${}^{rp}\text{Fifo}$.

$$\text{Fifo} \parallel U_n \xrightarrow{in}_r \text{Fifo}(1) \parallel (U_{n-1} \parallel_{\{\omega\}} \underline{out}.\underline{\omega}) \xrightarrow{\mathbb{A}}_r P' = \underline{\text{Fifo}}(1) \parallel (U_{n-1} \parallel_{\{\omega\}} \underline{out}.\underline{\omega})$$

Here, $\underline{\text{Fifo}}(1) = (\underline{in}.\text{Fifo}(2) + \underline{out}.\text{Fifo}(0))$ can perform $\xrightarrow{\emptyset}_r$ to itself; but by the refusal semantics we could have $P' \xrightarrow{1}$ only if $(U_{n-1} \parallel_{\{\omega\}} \underline{out}.\underline{\omega})$ is able to refuse both *in* and *out*. And this is clearly not the case. We are currently working on this qualitative/quantitative issue by defining a slight variation of the faster than preorder as given in [6] to relate processes w.r.t. the restricted class of tests \mathcal{U} as in [4] but by some variant of refusal trace inclusion.

Our aim is to tune **FASE** to allow the analysis of larger systems, where the performance module needs more attention since it implements the theories introduced above. A first important result, we have already obtained, is the improvement of the catastrophic-cycles detection; ensuring their absence is the basis for any further performance analysis. A second result regards the calculation of the bad cycle, especially when we consider complex processes. However, the graph G' used in Karp's algorithm could be very large, and we will investigate ways to minimise it. We are also working on a good strategy to determine the response performance of P for a given n . Different approaches are under investigation but they still need to be validated. Currently, **FASE** executes an exhaustive search on $\text{rRTS}(P)$ that looks for the n -critical path whose duration is maximal; clearly as n increases this solution becomes soon intractable, especially for complex processes. Even though it is a rough solution, at least it helped to validate the results on response performance presented in the above propositions.

Anyhow, **FASE** represents a good first step towards the creation of an integrated framework for the analysis of concurrent systems modelled through PAFAS. The improvements introduced with **FASE** and the possibility to derive the complete set of behavioural traces of the modelled system allowed us to study and validate many results, such as the ones stated in this paper, that would have been harder to calculate without an automated tool like **FASE**.

References

- [1] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.
- [2] Stephen D. Brookes, C. A. R. Hoare, A. W. Roscoe. *A Theory of Communicating Sequential Processes*. J. ACM, 31:560-599, 1984.
- [3] F. Corradini, M. R. Di Berardini and W. Vogler. *PAFAS at Work: Comparing the Worst-Case Efficiency of Three Buffer Implementations*. In Y.T. Yu and T.Y. Chen, editors, 2nd Asia-Pacific Conference on Quality Software APAQS 2001, IEEE, 231-240, 2001.
- [4] F. Corradini and W. Vogler. *Measuring the performance of asynchronous systems with PAFAS*. Theor. Comput. Sci, 335(2-3):187-213, 2005.
- [5] F. Corradini and W. Vogler. *Performance of pipelined asynchronous systems*. J. Logic and Algebraic Programming, 70:201-221, 2007.
- [6] F. Corradini, W. Vogler and L. Jenner. *Comparing the worst-case efficiency of asynchronous systems with PAFAS*. Acta Informatica, 38(11):735-792, 2002.
- [7] R. De Nicola and M.C.B. Hennessy. *Testing equivalence for processes*. Theoretical Comput. Sci., 34:83-133, 1984.
- [8] R.M. Karp. *A characterization of the minimum cycle mean in a digraph*. Discrete mathematics 23(3):309-311, 1978.
- [9] Gerwin Klein. *Jflex user's manual*. <http://jflex.de/>, 2001.
- [10] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [11] Mark P. Jones. *jacc: just another compiler compiler for Java. A Reference Manual and User Guide*. <http://web.cecs.pdx.edu/~mpj/jacc>, 2004.